A Dynamic Case-Based Planning System for Space Station Application*

F. Oppacher
D. Deugo

School of Computer Science,
Carleton University, Ottawa, Ontario,
Canada K1S 5B6.

Abstract

We are currently investigating the use of a case-based reasoning approach to develop a dynamic planning system. The dynamic planning system - DPS for short - is designed to perform resource management, i.e. to efficiently schedule tasks both with and without failed components. Our approach deviates from related work on scheduling and on planning in AI in several respects. In particular, we attempt to equip the planner with an ability to cope with a changing environment by dynamic replanning, to handle resource constraints and feedback, and to achieve some robustness and autonomy through plan learning by dynamic memory techniques. We briefly describe the proposed architecture of DPS and its four major components: the PLANNER, the plan EXECUTOR, the dynamic REPLANNER, and the plan EVALUATOR. The planner, which is implemented in Smalltalk, is being evaluated for use in connection with the Space Station Mobile Service System (MSS).

1. Introduction

In real-world planning tasks it is often necessary to manage plans that contribute to more than one goal, to flexibly adjust plans that conflict with concurrent goals, and to anticipate and avoid bad planning. Moreover, to achieve some degree of autonomy, a planning system that may have to operate in changing environments must rely on feedback. The presence of feedback presupposes an ability for dynamic replanning, i.e. for reacting to changing conditions as execution proceeds. Feedback tasks also often involve tight time and other resource constraints.

We have argued in [Deugo et al. 88] that the feedback-imposed needs for dynamic replanning and for dealing with continuous resources like power or time are beyond the capabilities of traditional schedulers such as, e.g., PERT and CPM [Moder and Phillips 70], and AI planners such as, e.g., STRIPS [Nilsson 80] and NOAH [Rich 83]. For example, a STRIPS-like approach seems to presuppose that the system's world model is and remains correct, that the operator always does exactly what is required, that nothing happens between making the plan and executing it, and that the plan is executed precisely as planned. In short, such an approach works only in static situations and not in more realistic settings. Postponed commitment planners [Stefik 81] attempt to solve this problem by postponing the commitment of the exact order in which their task are to be

* This research is undertaken on behalf of the Department of Communication, Communications Research Centre, with funding provided by the Canadian Space Station Program Office of the National Research Council.

executed until execution time. However, the size of the partially ordered set of tasks constructed can be exponential in size to the set of tasks, and may not account for every type of situation.

Realistic planners should also be able to adapt old plans to the current situation and to extend their plan library by learning. A robust and efficient planner should neither be forced to give up if there is no appropriate, ready-made plan in its library nor have to replan always from scratch.

We believe that a combination of dynamic memory techniques [Kolodner 84] [Schank 82] and case-based reasoning techniques [Hammond 86] [Kolodner et al. 85] is necessary to successfully tackle all of these problems. Section 2 outlines how the architecture of the DPS integrates several dynamic memory and case-based techniques and identifies the relationships among its major components. Section 3 describes these components in more detail, and section 4 proposes extensions to our approach and summarizes.

2. DPS Architecture

The tasks to be planned for by the DPS depend on the availability of different resources such as time and power. A human operator enters information about resource availability in the form of initial plan constraints. The DPS relies on a feedback loop to provide information about the success or failure of the plan's execution. Subsequent planning sessions involving this or a similar plan can use this information and thereby gain from past experience. Thus, starting with an initial library of plans, the DPS acquires new plans through a form of plan learning from past plans.

Our dynamic memory and case-based approach to the planning problem postulates four major components: the PLANNER, the plan EXECUTOR, the dynamic REPLANNER, and the plan EVALUATOR. Figure 1 shows how these components fit together.

The PLANNER controls the planning process, from information input, past plan locating, to plan construction. Initially the operator configures the planner with its resource information. This provides the resources the planner can use over the plan execution period. Next, the operator enters the planning parameters, i.e. the tasks and constraints, to help set up the plan construction phase. The Locator uses a case-based approach to locate a past similar plan-goal-resources configuration. Using the supplied input information, the Locator indexes into a library of old plans, indexed by their goals (tasks), in an attempt to find a plan that matches the current planning parameters. If a matching plan can be found, it is passed to the plan EXECUTOR component. If no plan can be found, the Locator attempts to find a plan whose goals are a subset or superset or generalizations or specializations of the current goals. This past plan is then modified by the Constructor's domain planning information or planning heuristics found in the knowledge base, or by the operator, to create a new plan to be executed by the plan EXECUTOR component. The index to the knowledge about planning and plan modification is formed using the task and constraint information about the task the planner is currently considering. The modified plan is then verified using expected resource information to insure the plan's integrity. A failure in a task's verification will cause the Constructor to alter the plan.

This approach enables the planner to continue planning even though it has no exact ready-made plan to deal with the current tasks.

The Explainer collects all the planning activity information and can then explain to the operator what the plan is and how it was constructed. In addition to this, it can also use past plan exception information to describe failure conditions that could arise in the execution of the plan. This information is also used by the Constructor to help build better plans, or by the operator to help

alter the Constructor's proposed plan. Figure 2 provides an example of the planning activity information stored by the Constructor when an existing plan is modified to meet the current goals.

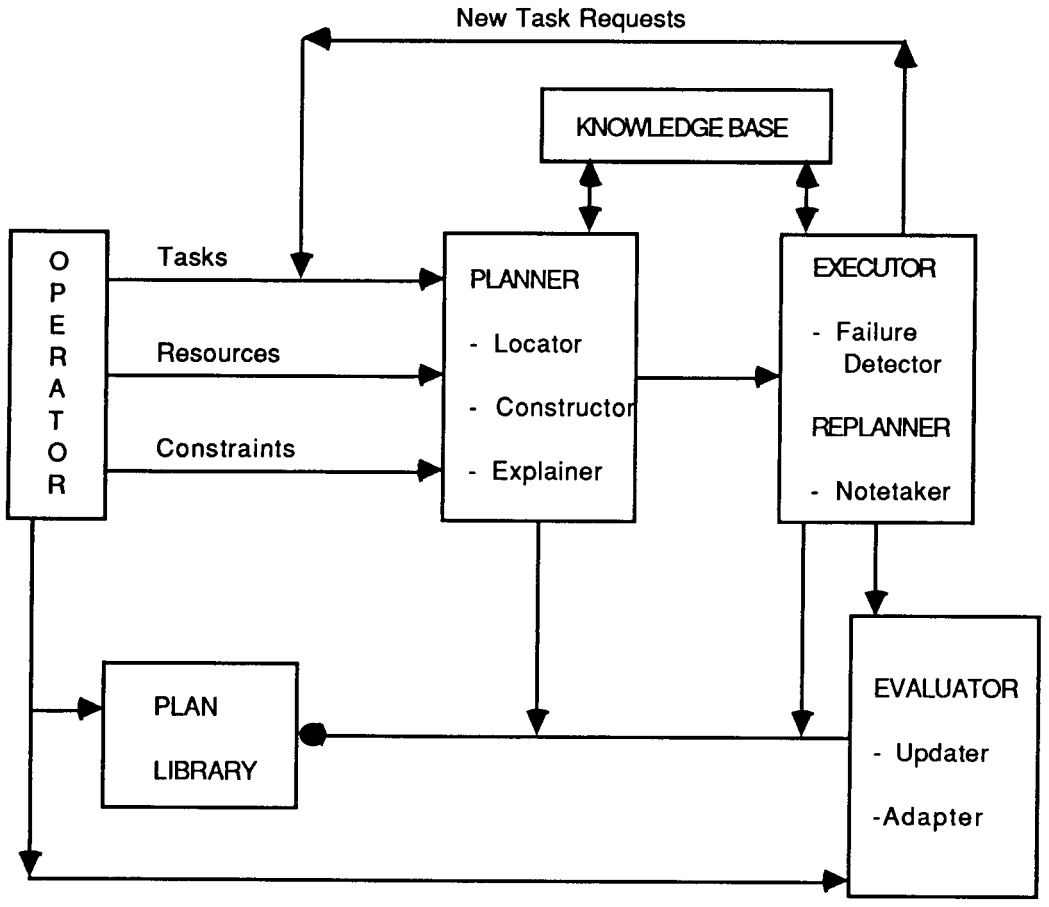

Figure 1. Architecture of the DPS

The plan EXECUTOR component takes the plan and starts the execution of it. If the Failure Detector experiences a failure condition (lack of resource) that was unforeseeable at the time of plan construction and arose during the plan's execution, the exception is noted by the Notetaker, and the dynamic REPLANNER component is activated. This component will attempt to reorder the plan, remove failing goals, or ask the operator for assistance in order to keep the EXECUTOR's plan execution continuing. These actions are found in the knowledge base and are indexed like any other planning information. After all, when a plan fails one does not want to stop the execution since resources have been allocated and are ready to use. The information about what replanning was done is noted by the Notetaker for later use by the EVALUATOR component.

The dynamic REPLANNER component is an important improvement that distinguishes our planner from other planners described in the literature. It prevents minor faults from stopping plan execution, and causes only moderate modifications of the plan. The REPLANNER uses the planning technique known as goal planning or reactive planning [Schoppers 87] to keep a plan executing. It also indexes into old plans to see if any replanning information is available for use in the current situation.

[1] Plan B selected with a rating of one extra task, one missing task, and no failures.
[2] Adding missing tasks, to Plan B.
[3] Removing extra tasks, from Plan B.
[4] Verifying Plan B.
[5] Inspection task failed, replanning.
[6] Rule 'Move task to end of Plan' fired, executing action rule.
[7] Verifying Plan B.
[8] Plan verified, ready for execution.

Figure 2. Explainer's Planning Activity Information

After the plan has executed, the plan and the information provided by the Notetaker, i.e. exceptions and replanning descriptions, are given to the EVALUATOR component. If the plan was an old plan that executed successfully, this information is added to the plan in the plan library to provide added support information for it. If the plan was newly created and it executed successfully, it is added to the plan library along with the goals it satisfied. If the plan failed, the exceptions and replanning information are recorded in the plan along with the reasons why the goal, or goals, failed. All of these transactions are handled by the Updater. If the plan has had a bad track record, it may also be altered by the Updater using the Notetaker information to make it a 'better plan' in the future. An updated plan is 'better' than the original plan because either tasks with a proven history of failure have been removed from it or it includes Notetaker information which can be used in future planning sessions. Failure and success information are valuable in determining the best plan for the current situation. The operator is also part of this activity: he/she helps to verify the reasoning of the Updater, and ensures the sanity of updates for the plan library.

The EVALUATOR component helps the planner acquire new plans and knowledge by learning from itself. This is achieved by adding new, successfully executed plans constructed by the Constructor, and by altering old plans due to planning failures. By recording the failures, the EVALUATOR also learns to fix and adapt plans to new environments over time.

The PLANNER, REPLANNER, and EVALUATOR components rely on dynamic memory and case-based techniques to generate a plan, to alter a plan due to a failure, to store new plans, and to update old ones. These techniques enable the DPS to work in a dynamic environment and be ready to meet a wide range of unforeseen changes.

3. Component Definitions

The inputs to PLANNER consist of the plan resources, the tasks to be planned, and the constraints on the tasks. In our prototype implementation this information is provided by a human operator. The output of the PLANNER is a plan which is later executed by the EXECUTOR and updated or added to the plan library. We now briefly discuss each part.

Resources can take the form of any type of (usually time-varying) physical supply, such as electrical power. In some cases, the consumption of a resource can increase or decrease the supply of another. This is know as a Supplied resource. The planner must keep track of all available resources. Resources are defined by resource functions that enable the planner to predict the supplies at time t.

Tasks are treated by the current implementation as unit activities that cannot be further decomposed. The DPS schedules tasks but does not plan for the execution of individual tasks. They are assumed to be directly executable by the EXECUTOR; future extensions will allow the

PLANNER to be applied to tasks as well, thereby facilitating the construction of hierarchical plans. A task's information aids the planner to efficiently position it among the other tasks in the plan. This information is entered by the operator using a form-based approach. A task form with data slots identifying the task, its constraints, its requirements, and the supply of different resources it increases, is provided for the operator to fill in. Thus, a task has the following structure:

```
(task-name-slot                 ; the name of the task; e.g., operation X.
activity-slot                   ; what the task is to do; e.g., running operation X.
constraint-1-slot               ; e.g., must be done by 0800 hours.
   •
   •
constraint-n-slot
resource-required-1-slot        ; e.g., operation X uses 50 watt hours of electricity.

   •
   •
resource-required-n-slot
resource-supplied-1-slot        ; e.g., operation X raises the temperature by 1°C.
   •
   •
resource-supplied-n-slot).
```

Constraints are identical to task constraints, except that they constrain a plan. For example, a plan constraint could be that the plan must start execution before 0800 hours and finish execution by 0900 hours. The operator supplies this information to the planner by entering the data on a plan constraint form.

A **Plan** is an ordered sequence of tasks, produced by the PLANNER using the initial planning information, for execution by the EXECUTOR. Each task in a plan has two new slots added to it, a start-time-slot and finish-time-slot, which are used by the EXECUTOR to determine when each task is to start and finish execution. A plan has the following structure:

```
(plan-name-slot                 ; the name of the plan.
success-slot                    ; a count of the number of times, initially zero, the plan
                                  has executed successfully without having to be
                                  replanned by the Replanner.
failure-1-slot                  ; failure slots include information about how the plan
                                  failed and what was done to correct it.

failure-2-slot
   "
   "
failure-n-slot
task-1-slot                     ; points to component task; e.g., operation X.
task-2-slot
   "
   "
task-n-slot).
```

The **Plan Library**, as the name suggests, is a library of past plans that have either been created initially by the operator and entered into the library, or have been created by the system and added to the library. They are stored sequentially and are indexed by the search mechanisms of the Constructor component of the PLANNER. The library is memory-resident in the current

implementation but will be converted to a database management system when issues such as size, access, and information updating become important.

The **PLANNER** develops a plan to execute the operator-entered tasks. It uses past plans, stored in the plan library, in its attempt to locate or build a new plan. The PLANNER consists of three components: the Locator, the Constructor, and the Explainer.

The **Locator** takes all of the tasks' activity-slot identifiers and tries to locate a past plan that contains only those tasks. If such a plan is found in the plan library, it is passed to the Constructor. If no plan can be found, the Locator looks for a past plan whose tasks properly include the current requirements. If such a plan can be found, it is passed to the Constructor with the tasks in excess of the current requirements marked. If still no plan can be found, the Locator looks for a past plan that contains the maximum subset of tasks currently required. This plan is passed to the Constructor with the tasks missing from the plan identified. When multiple plans are located, the one with the best success rate, calculated by subtracting its successes from its failures, is returned.

The **Constructor** first checks to see if the plan contains only the required tasks. Any excess tasks are simply removed. If it has less, the Constructor rules stored in the knowledge base are accessed to decide what action to take. Specific Constructor rules have both a task and a constraint as rule identifiers, and are considered first. If there are no such rules, more general rules are accessed that rely only on the task or the constraints, but not both. The actions taken by such rules include: to append a task to the end of the plan, to put it at the front of the plan, to put it after a specific task in the plan, or to find the first available position that satisfies its constraints.

Using the resource function information, the plan is verified to ensure that all task and plan constraints are met. If a constraint fails, the combination of constraint failure and task is used as an index to a rule which provides the appropriate action to be taken with the plan. Such actions could take the form of: removing the task, delaying it until its resource requirements are met, or asking the operator for help. These constraint rules depend on the type of constraints and tasks handled. Once the plan is constructed, it can be verified or altered by the operator if desired, and then passed to the Executor.

The **Explainer** component describes what actions were taken to create the plan. It identifies what and why past plans were chosen, the problems and successes the past plan had, what actions were done in creating the current plan from the past plan, and what constraint problems were found and solved for the plan. The Explainer can be turned on during plan generation to allow the operator to view the creation process of the plan. Alternatively, the operator can ask individual questions at the end of the planning phase.

The **EXECUTOR**, using the planning information, executes the plan. It sequentially takes each task in the plan and performs that task's activity. A task finishes when its task activity ends or when its finish-time is reached. Information about the execution of the plan is stored by the Notetaker. At plan completion, the plan and its execution information are passed to the EVALUATOR.

For implementation purposes, a plan executes in discrete time slices. At each time slice, the operator can view the current state of the plan, the task executing, and the resource information. He can also vary the resource information in this period to cause a resource failure of the task, thereby forcing the replanning mechanism.

The **Failure Detector** monitors the resource sensors to ensure that none of a task's constraints are being violated. If at any time a task constraint is violated, the violation is noted by the Notetaker, the plan execution is stopped, and control is passed to the REPLANNER. The

REPLANNER will produce an adapted plan and return it to the EXECUTOR to restart execution from the point of interruption.

The **REPLANNER** first checks to see whether a given failure has been detected before. This is done by looking in the past plan on which the current plan is based. If it had failed in a similar manner before, the replanning information stored in the past plan is used to adapt the current plan, and the newly adapted plan is passed back to the EXECUTOR to commence execution. If there is no replanning information, the REPLANNER consults its replanning knowledge base. It uses the task and failing constraint to index a replanning rule to adapt the current plan. These rules are a subset of the Constructor rules and have the same form. Their actions may consist of deleting the task, delaying it, repositioning it, stopping the plan's execution altogether, or asking the operator for help. The actions taken by the REPLANNER are recorded by the Notetaker before the adapted plan is passed back to the EXECUTOR.

The **Notetaker** is responsible for recording all information about the execution of a plan, in particular its successes and failures. In the case of failures, the Notetaker records the failure causes and any replanning information. This includes information about successful replanning episodes and the replanning rules used by them. Thus, the Notetaker fills in the following information slots:

Success-slot - Initially true. If a plan fails, it is set to false.
Number of failures - Initially zero, incremented by one for each failure.
Failure-slot - Initially empty. One slot is created and filled in per failure type. A task's failure-slot identifies the task executing, the failure-type, the failure-cause, the replan-type, and the failure-count.

The **EVALUATOR** receives the plan and the Notetaker information about it, and must decide what actions should be taken with the plan. It has several options depending on the plan type and the success or failure information. These options include:

Success of Old Plan - If the plan was previously used, its success-slot is incremented to boost its strength.
Success of New Plan - If the plan is a new one and it executed successfully, it will be added to the plan library with its success-slot set to one.
Failure of New Plan - If the plan had a minor failure (for example, one task out of fifty was delayed), the plan may be added to the library with a failure-slot filled in. If the failure has occurred often, the plan is discarded as being invalid.
Failure of Old Plan - If a task in a plan failed for the first time, the failure information is added to a new failure-slot for that task in the plan library. If a failure of this type already exists in a task's failure-slot, the failure-slot-count is incremented.

Just as the Constructor 'massages' the retrieved plan's constraints and tasks to make them fit the current situation, so too does the EVALUATOR. Using knowledge base rules, it massages the failure conditions so that they are adapted and appropriate for the current plan and its situation.

The **Updater** has the job of updating or adding a new plan or plan information to the plan library. Its algorithm is based on the four possible types of updates identified by the EVALUATOR. Updates can be monitored and, if it is desired to maintain close control over the reasonableness of the evolving plan library, modified by the operator.

The **Adapter** reorganizes plans that have failed often to improve them for future use. It uses replanning information in the failure-slots to determine whether to remove a task from the plan, delay the task in the plan, reorder the task, or remove the plan from the library. The Adapter will be activated when plan failures have reached a preset threshold.

Another part of its task is to look for several plans that can be generalized into a single abstract plan that preserves all of the information of the other plans. The latter can then be removed from the library. This reduction of the number of plans improves the search efficiency without compromising the planning knowledge in the library. It also helps prevent the library from filling up with many similar plans and provides plans that can be used in many different situations. We intend eventually to apply this process of generalization in the context of problem solving to the rules used by the Constructor and Replanner, thus keeping redundant rules from cluttering the knowledge base.

4. Summary

As can be seen from this brief description, our planner is concerned with many of the different areas of case-based reasoning. We retrieve past cases based on the number and type of tasks in the current situation they match, and use the number of times a plan succeeds in similar situations. We use a knowledge base of rules to aid in the plan transformation of a past plan to the current situation. We use past planning information and current planning operations to explain the planning task. We do dynamic replanning using the same knowledge base to keep the plan executing. We note all of this information and encode it into a past plan or new plan, enabling plan cases to be better utilized on the next planning iteration. The feedback loop enables plans and new plan learning to evolve with the environment over time.

By using dynamic memory and case-based reasoning techniques, combined with knowledge based techniques for replanning, we have presented a design that handles resource constraints, feedback, and achieves both robustness and some degree of autonomy through plan learning. Although not essential to any part, the operator can guide the overall planning process and control the acquisition of new plans and rules for replanning. With the current design and the future enhancements, we feel we are approaching a realistic, efficient planner.

Planned enhancements to the DPS not already mentioned include improving the process of unifying the supplied operator planning information to that of a stored plan, unifying all of the planning information, not just the goals, and when the supplied information is incomplete. Also, the DPS will be enhanced to recognize dangers and opportunities during the plan's execution. This will allow the plan to benefit from or avoid problems during execution in its current environment. A final enhancement will be to take the plan's failure information and generalize it into an action recommendation. This recommendation is used in replanning before the specific task/constraint failure information is accessed. By doing this, additional planning failures should be avoided after replanning because the recommendation has already taken them into account.

5. Acknowledgments

We wish to thank Prof. D. Thomas of the School of Computer Science, Carleton University, P.J. Adamovits and R.A. Millar of the Department of Communication, Communications Research Centre, and the the Canadian Space Station Project Office for supporting this work.

6. References

Deugo, D. L., Oppacher, F., Thomas, D., Planning Techniques Survey: Their Applicability to the Mobile Service System, TR/SCS, Carleton University, Ottawa, 1988.

Hammond, K. J., CHEF: A Model of Case-based Planning, AAAI 1986, pp. 267-271.

Kolodner, J. L., Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model, Lawrence Erlbaum Associates, Publishers, 1984.

Kolodner, J. L., Simpson, R. L., Sycara-Cyranski K. L., A Process Model of Case-Based Reasoning in Problem Solving, IJCAI 1985, pp. 284-290.

Moder, J. J., Phillips, C. R., Project Management with CPM and PERT, Van Nostrand Reinhold Company, 1970.

Nilsson, Nils J., Principles of Artificial Intelligence, Palo Alto, California Tioga Press, 1980, pp. 275-360.

Rich, E., Artificial Intelligence, McGraw-Hill Book Company, 1983, pp. 247-294.

Schank, R. C., Dynamic Memory: A Theory of Reminding and Learning in Computers and People, Cambridge University Press, 1982.

Schoppers, M. J., Universal Plans for Reactive Robots, IJCA 1987, 1039-1046.

Stefik, M., Planning with Constraints (MOLGEN: Part 1), Artificial Intelligence, 16, 2, May 1981, pp. 111-140.